Lecture 2: Implementing ADTs

Data Structures and Algorithms

Warm Up – Discuss with your neighbors!

From last lecture:

- What is an ADT?
- What is a data structure?

From CSE 143:

- What is a "linked list" and what operations is it best at?
- What is a "stack" and what operations is it best at?
- What is a "queue" and what operations is it best at?

Socrative:

www.socrative.com

Room Name: CSE373

Please enter your name as: Last, First

Announcements/ Questions

No overloading, wait for drops

Class page to be live tonight

Sections start tomorrow

TA Introductions!

Ryan Pham
Office Hours: Monday 9:30-11:30
Section: Thursday 1:30

Meredith Wu
Office Hours: Friday 1:00 – 3:00pm
Section: Thursday 10:30

Design Decisions

For every ADT there are lots of different ways to implement them

Example: List can be implemented with an Array or a LinkedList

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!

> A common topic in interview questions

Review: "Big Oh"

efficiency: measure of computing resources used by code.

- can be relative to speed (time), memory (space), etc.
- most commonly refers to run time

Assume the following:

- Any single Java statement takes same amount of time to run.
- A method call's runtime is measured by the total of the statements inside the method's body.
- A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

We measure runtime in proportion to the input data size, N.

- growth rate: Change in runtime as N gets bigger. How does this algorithm perform with larger and larger sets of data?

Say an algorithm runs $0.4N^3 + 25N^2 + 8N + 17$ statements.

- We ignore constants like 25 because they are tiny next to N.
- The highest-order term (N^3) dominates the overall runtime.
- We say that this algorithm runs "on the order of" N^3.
- or **$O(N^3)$** for short ("**Big-Oh** of N cubed")

Review: Complexity Class

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	$O(N)$	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion

Review: Case Study: The List ADT

list: stores an ordered sequence of information.

- Each item is accessible by an index.
- Lists have a variable size as items can be added and removed

Supported Operations:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list

List ADT tradeoffs

Time needed to access i-th element:

- Array: O(1) constant time
- LinkedList: O(n) linear time

Time needed to insert at i-th element

- Array: O(n) linear time
- LinkedList: O(n) linear time

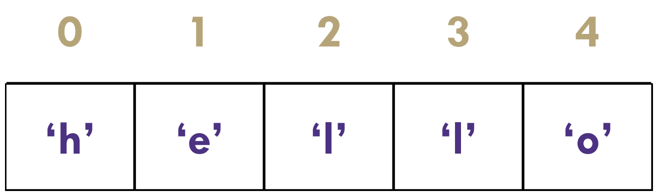
Amount of space used overall

- Array: sometimes wasted space
- LinkedList: compact

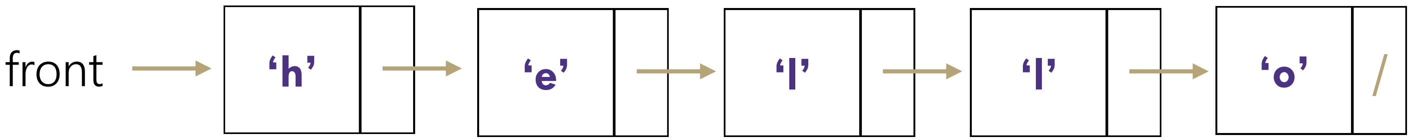
Amount of space used per element

- Array: minimal
- LinkedList: tiny extra

```
char[] myArr = new char[5]
```




```
LinkedList<Character> myLl = new LinkedList<Character>();
```



Thought Experiment

Discuss with your neighbors: How would you implement the List ADT for each of the following situations? For each consider the most important functions to optimize.

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

LinkedList

Situation #2: Write a data structure that implements the List ADT that will be used to store the count of students who attend class each day of lecture.

ArrayList

Situation #3: Write a data structure that implements the List ADT that will be used to store the set of operations a user does on a document so another developer can implement the undo function.

Stack

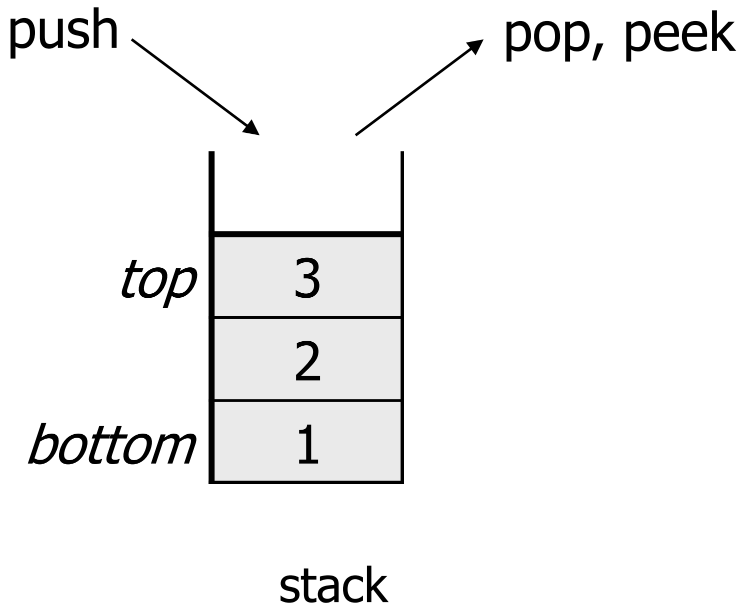
Review: What is a Stack?

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").

basic stack operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size()**: how many items are in the stack?
- **isEmpty()**: true if there are 1 or more items in stack, false otherwise



Implementing a Stack with an Array

push(3)
push(4)
pop()
push(5)

0	1	2	3	4
3	5			

`numberOfItems` = $\boxed{2}$

Review: Generics

```
// a parameterized (generic) class
public class name<TypeParameter> {
    ...
}
```

- Forces any client that constructs your object to supply a type.
  - Don't write an actual type such as String; the client does that.
  - Instead, write a type variable name such as `E` (for "element") or `T` (for "type").
  - You can require multiple type parameters separated by commas.
- The rest of your class's code can refer to that type by name.

```
public class Box {
   private Object object;
   public void set(Object object) {
      this.object = object;
   }
   public Object get() {
      return object;
   }
}
```

```
public class Box<T> {
   private T t;
   public void set(T t) {
      this.t = t;
   }
   public T get() {
      return t;
   }
}
```

More details: https://docs.oracle.com/javase/tutorial/java/generics/types.html

Implementing a Generic Stack

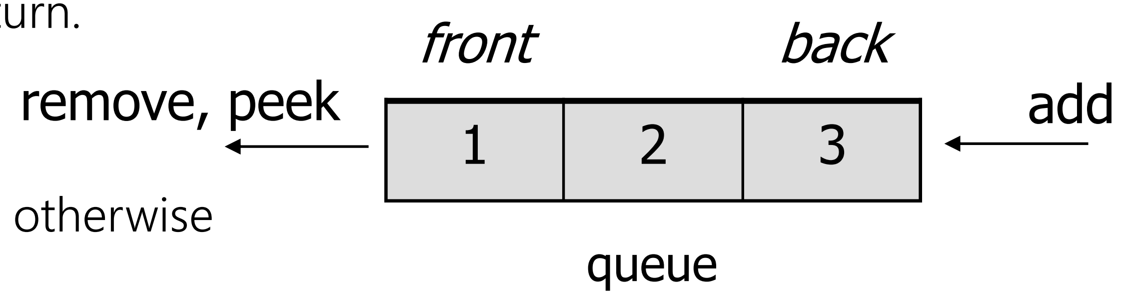
Review: What is a Queue?

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

basic queue operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek():** Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise



Implementing a Queue

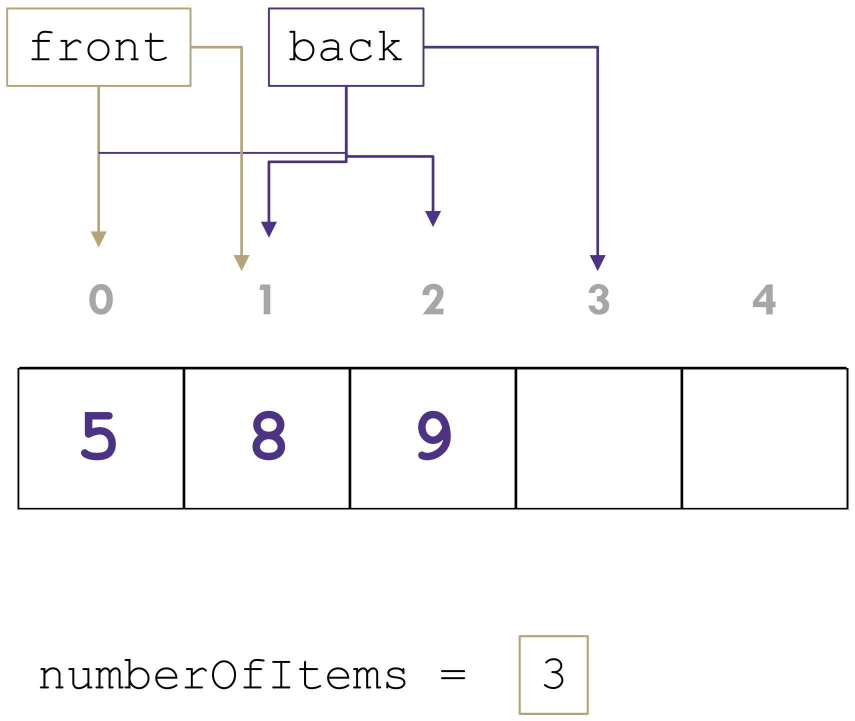
enqueue(5)
enqueue(8)
enqueue(9)
dequeue()

0	1	2	3	4
5	8	9		

`numberOfItems = 3`

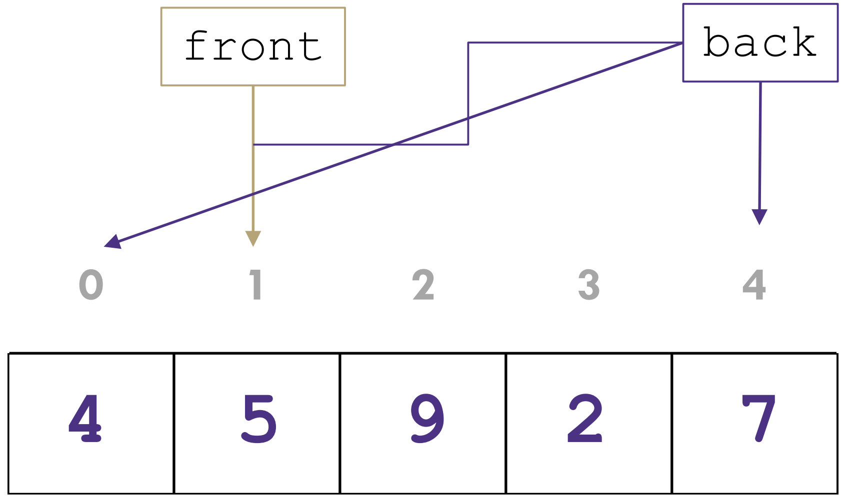
Circular Queues

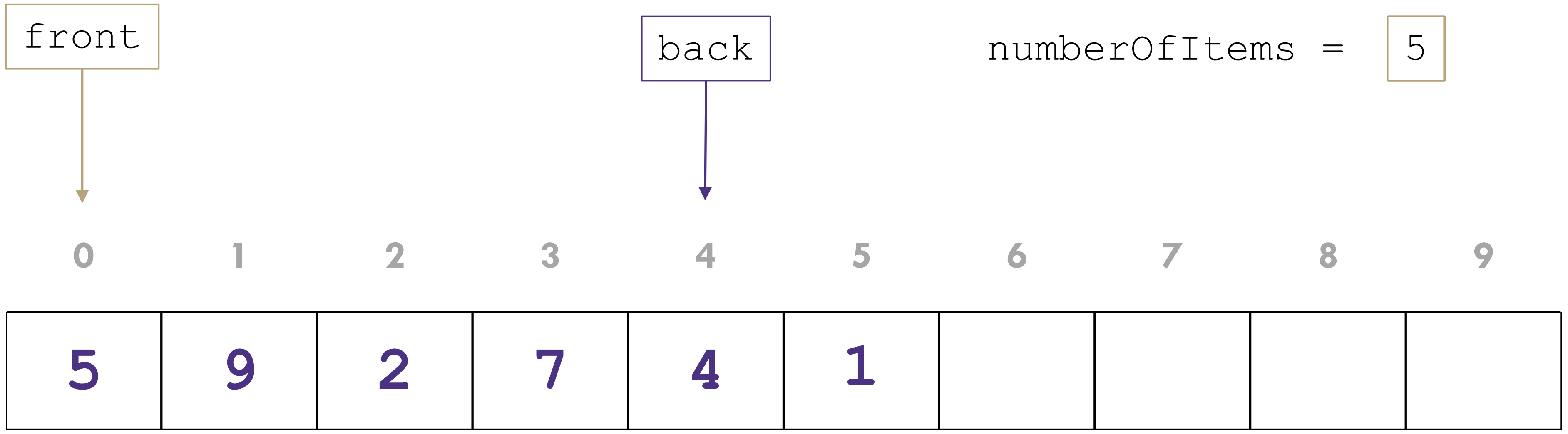
enqueue(5)
enqueue(8)
enqueue(9)
dequeue()



Wrapping Around

enqueue(7)
enqueue(4)
enqueue(1)





TODO list

Fill out survey!
- Link on class page

Class webpage to be live tonight:
- Skim through full Syllabus on class web page
- Sign up for Piazza
- Review 142/143 materials. Materials provided on class webpage